Performance Modeling and Tuning of an Unstructured Mesh CFD Application

William D. Gropp,[*] Dinesh K. Kaushik,[†] David E. Keyes,[‡] Barry F. Smith[§]

Abstract

This paper describes performance tuning experiences with a three-dimensional unstructured grid Euler flow code from NASA, which we have reimplemented in the PETSc framework and ported to several large-scale machines, including the ASCI Red and Blue Pacific machines, the SGI Origin, the Cray T3E, and Beowulf clusters. The code achieves a respectable level of performance for sparse problems, typical of scientific and engineering codes based on partial differential equations, and scales well up to thousands of processors. Since the gap between CPU speed and memory access rate is widening, the code is analyzed from a memory-centric perspective (in contrast to traditional flop-orientation) to understand its sequential and parallel performance. Performance tuning is approached on three fronts: data layouts to enhance locality of reference, algorithmic parameters, and parallel programming model. This effort was guided partly by some simple performance models developed for the sparse matrix-vector product operation.

1 Introduction

Our application is typical of a large number of high-performance applications arising from systems of nonlinear partial differential equations with multiple time scales discretized through finite-volume, finite-element, or finite-difference techniques on unstructured grids. Such problems are challenging for today's high-end cache-based architectures because they tend to be memory-bandwidth limited in some phases and instruction scheduling limited in other phases, as a result of their relatively high algorithmic efficiency. This high algorithmic efficiency leads to low computational intensity (floating-point operations per data reference) that stresses hardware that was designed for flattering dense linear algebra benchmarks (that generally require $O(n^2)$ memory references for $O(n^3)$ floating-point operations) . On the other hand, such problems scale well in processor number because of their regular and easily schedulable high concurrency and favorable surface-to-volume scaling of communication to computation for domain-based data decompositions.

[*] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, `gropp@mcs.anl.gov`. This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

[†] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, and Computer Science Department, Old Dominion University, Norfolk, VA 23529, `kaushik@mcs.anl.gov`. This work was supported by a GAANN Fellowship from the U.S. Department of Education and by Argonne National Laboratory under contract 983572401.

[‡] Mathematics & Statistics Department, Old Dominion University, Norfolk, VA 23529, ISCR, Lawrence Livermore National Laboratory, Livermore, CA 94551, and ICASE, NASA Langley Research Center, Hampton, VA 23681, `keyes@icase.edu`. This work was supported by the National Science Foundation under grant ECS-9527169, by NASA under contracts NAS1-19480 and NAS1-97046, by Argonne National Laboratory under contract 982232402, and by Lawrence Livermore National Laboratory under subcontract B347882.

[§] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, `bsmith@mcs.anl.gov`. This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

1.1 Background of Computations

The NASA code FUN3D [1, 3], solves the Euler and Navier-Stokes equations of fluid flow in incompressible and compressible forms with second-order flux-limited characteristics-based convection schemes and Galerkin-type diffusion. FUN3D is used for design optimization of airplanes, automobiles, and submarines, with irregular meshes comprising as many as several million grid points. The optimization loop involves many analysis cycles. Thus, time to reach the steady-state solution in each analysis cycle is crucial to conducting the design optimization in a reasonable amount of time. Our effort has focused on achieving small time to convergence without compromising scalability, by means of appropriate algorithms and architecturally efficient data structures.

The solution algorithm is Newton-Krylov-Schwarz [11] with block-incomplete factorization on each subdomain of the Schwarz preconditioner and with varying degrees of overlap. For asymptotic scalability this algorithm requires a coarse grid preconditioning step; for our purpose, however, we do not need this step because the nonlinear stiffness of the problem (even in the absence of fluid-mechanical shocks) requires a timestepping globalization of Newton's method [15]. The diagonal dominance (though very slight at high Courant-Friedrich-Levy number (CFL) numbers, which eventually reach to 10^5) is sufficient to keep the conditioning of the linear Newton correction problems reasonable.

Our experiments in this paper are limited to incompressible Euler flows on unstructured tetrahedral grids about three-dimensional M6 wings of three sizes (22,677 vertices for a single processor, 357,900 vertices for tens to hundreds of processors, and 2.8 million vertices for thousand of processors).

1.2 Sample Parallel Performance Results

PETSc-FUN3D (that is, the discretization of FUN3D algorithmically reimplemented and performance tuned in the PETSc [4] framework) has run efficiently on many contemporary parallel machines and was awarded a 1999 Gordon Bell Prize in the "special" category at SC'99 [2].

We show some representative parallel scalability results in Figure 1 for a mesh with 2.8 million vertices running on up to 3072 nodes of ASCI Red. The implementation efficiency (parallel efficiency per time step) is 91% in going from 256 to 2048 nodes. For the data in Figure 1, we employed the `-procs 2` runtime option that enables two-processor-per-node multithreading during threadsafe, communication-free portions of the code. We have activated this feature for the floating-point-intensive flux computation subroutine alone because it does not suffer from memory bandwidth limitation [10]. On 2048 nodes, the resulting Gflop/s rate is 156, or 30% greater than for the single-threaded case on the same number of nodes. On 3072 nodes, the largest run we have been able to make on the unclassified side of the machine to date, the resulting Gflop/s rate is 227. Undoubtedly, further improvements to the algebraic solver portion of the code are also possible through multithreading, but the additional coding work does not seem justified at present.

Figure 2 shows a plot of aggregate flop/s performance and a log-log plot of execution time for the same grid of 2.8 million vertices on the three most capable machines to which we have thus far had access. In both plots of this figure, the dashed lines indicate ideal behavior. Note that although the ASCI Red flop/s rate scales nearly linearly, a higher fraction of the work is redundant at higher parallel granularities, so the execution time does not drop in exact proportion to the increase in flop/s. The number of vertices per processor ranges from about 22,000 to fewer than 1,000 over the range shown.

2 Performance Modeling and Tuning

In this section, we describe the details of the performance tuning process. Our approach is largely experimental, guided in part by the performance models that we have developed in [10]. We first present the data layouts that can reduce the number of cache misses, tend to use low memory bandwidth, and show better scalability, especially when the number of subdomains becomes large. Then we provide close to optimal

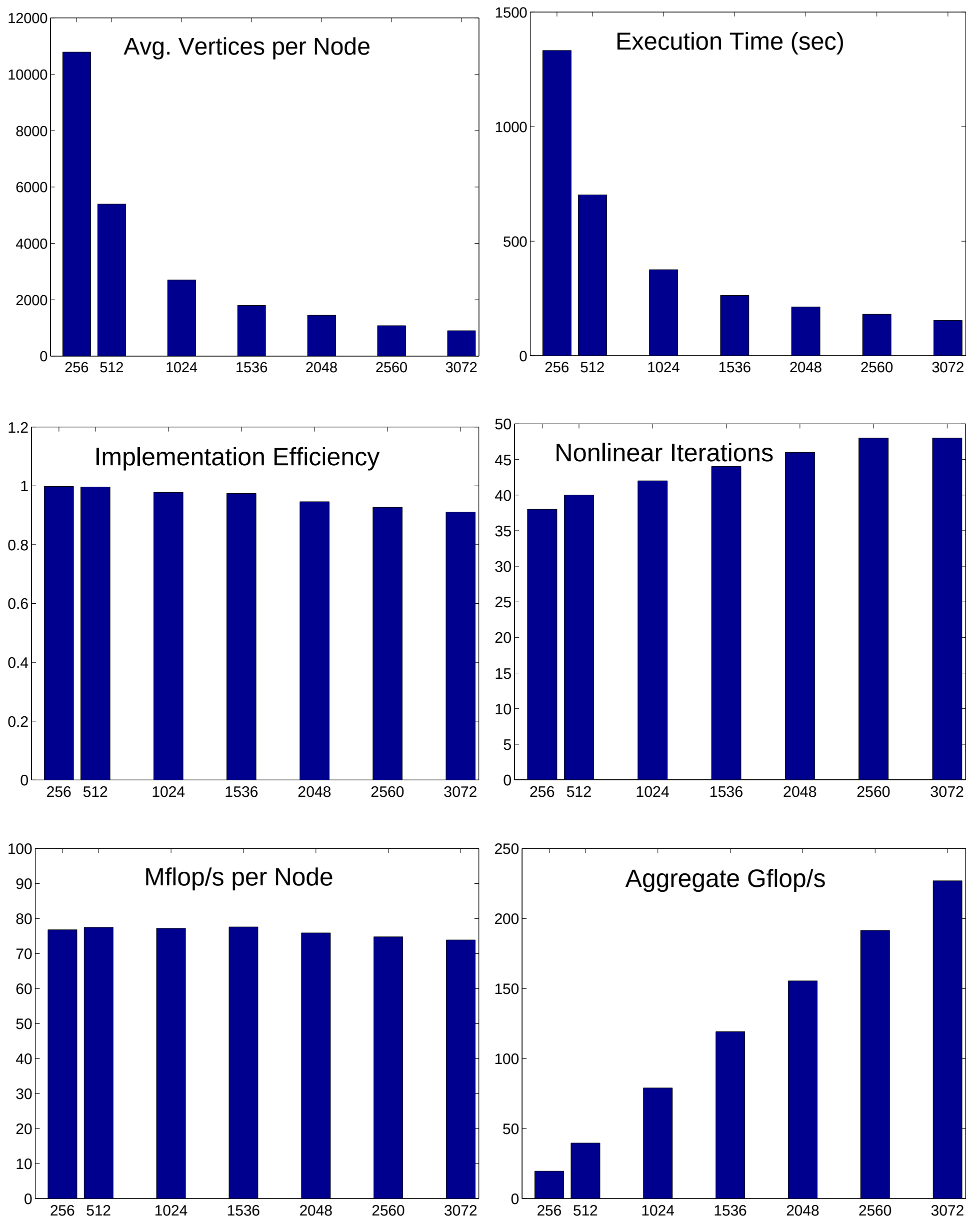
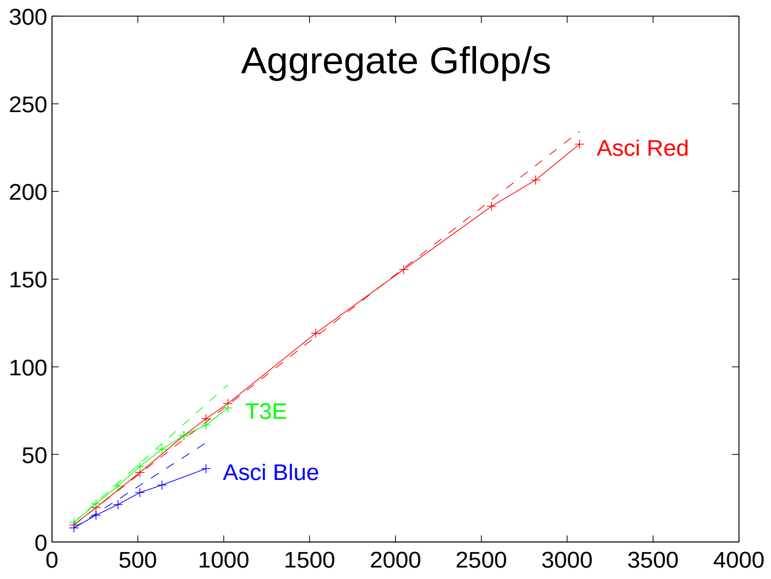


Figure 1: Average vertices of the mesh owned by each processor and five parallel performance metrics for a fixed-size problem on a 2.8 million vertex mesh, run on up to 3072 nodes of ASCI Red (each node consisting of two 333 MHz Pentium Pro processors). The preconditioner used in these results is block Jacobi with ILU(0) in each subdomain. However, we have now discovered that the block Jacobi with ILU(1) gives better execution times (see Table 4). We will update these results in our future work.



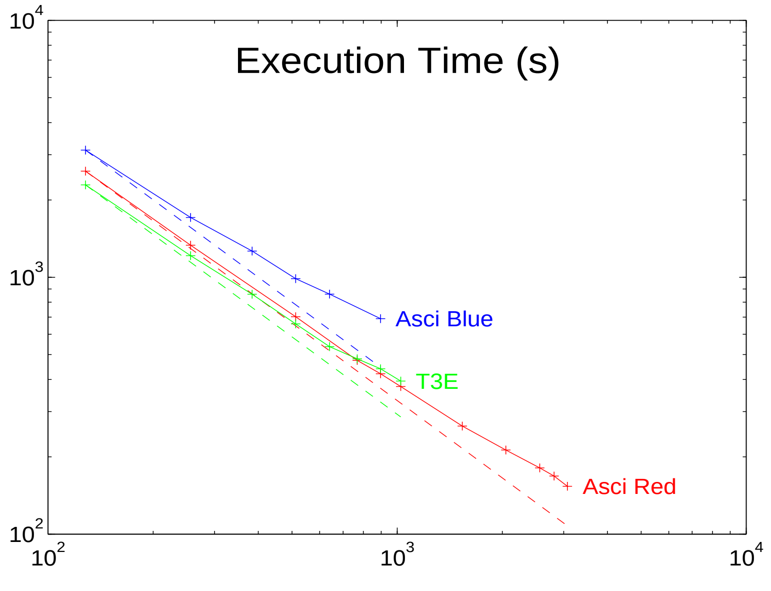


Figure 2: Gigaflop/s ratings and execution times on the ASCI Red, ASCI Pacific Blue, and Cray T3E for the 2.8 million vertex case, along with dashed lines indicating “perfect” scalings up to the available number of nodes up to (3,072 for ASCI Red).

Table 1: Execution times for Euler flow over M6 wing; fixed-size grid of 22,677 vertices (90,708 DOFs incompressible; 113,385 DOFs compressible); MIPS R10000, 250 MHz, cache: 32 KB data / 32 KB instruction / 4 MB L2. Activation of a layout enhancement is indicated by "$\times$" in the corresponding column. Improvement ratios are averages over the entire code; different subroutines benefit to different degrees.

Enhancements			Results			
Field	Structural	Edge	Incompressible		Compressible	
Interlacing	Blocking	Reordering	Time/Step	Ratio	Time/Step	Ratio
			83.6s	—	140.0s	—
×			36.1s	2.31	57.5s	2.44
×	×		29.0s	2.88	43.1s	3.25
		×	29.2s	2.86	59.1s	2.37
×		×	23.4s	3.57	35.7s	3.92
×	×	×	16.9s	4.96	24.5s	5.71

ranges for a large set of algorithmic parameters. In the end, we investigate the hybrid programming model on ASCI Red.

2.1 Reducing the Cache Misses

Since the gap between memory and CPU speeds is ever widening [12], it is crucial to utilize the data brought into the levels of memory hierarchy that are close to the CPU. To achieve this goal, the data structure storage patterns for primary (e.g., momenta and pressure) and auxiliary (e.g., geometry and constitutive parameter) fields should adapt to hierarchical memory. Three simple techniques (discussed below) have proved very useful in improving the performance of the FUN3D code, which was originally tuned for vector machines.

Table 1 shows the effectiveness of these techniques on one processor of the SGI Origin 2000. The combination of the three effects can enhance overall execution time by a factor of 5.7 (a table comparing several architectures is available in [14]). To further understand these results, we carried out hardware counter profiling on a R10000 processor. Figure 3 shows that edge reordering reduces the TLB misses by two orders of magnitude, while secondary cache misses (which are very expensive) are reduced by a factor of 3.5.

2.1.1 Interlacing

Interlacing creates the spatial locality for the data items needed successively in time. This is achieved by choosing

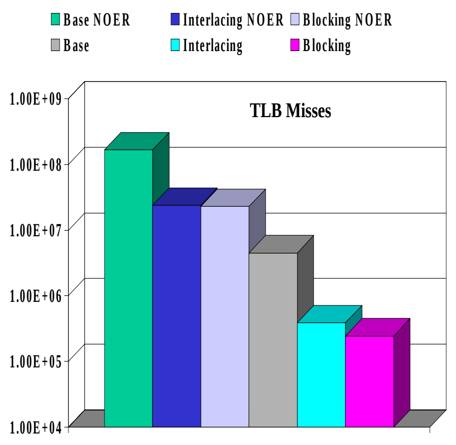
$$u1, v1, w1, p1, u2, v2, w2, p2, \ldots$$

in place of

$$u1, u2, \ldots, v1, v2, \ldots, w1, w2, \ldots, p1, p2, \ldots$$

for a calculation that uses u, v, w, p together. We denote the first ordering "interlaced" and the second "noninterlaced." The noninterlaced storage pattern is good for vector machines. For cache-based architectures, the interlaced storage pattern has many advantages: (1) it provides high reuse of data brought into the cache, (2) it makes the memory references closely spaced, which in turn reduces the translation look-aside buffer (TLB) misses, and (3) it decreases the size of the working set of the data cache(s), which reduces the number of conflict misses.

We illustrate these benefits on a simple kernel, the sparse matrix-vector product. We assume that the matrix of N rows is stored in CSR or compressed row format. Although the matrix is sparse, the vector it



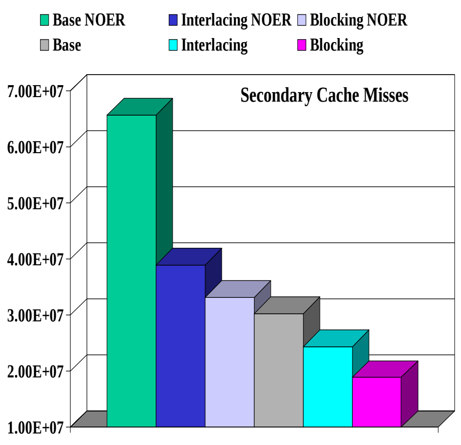


Figure 3: TLB misses (log scale) and secondary cache misses (linear scale) on one processor of an Origin 2000 for a 22,677 vertex case, showing dramatic improvements in data locality due to data ordering (grid edge reordering and field variable interlacing) and blocking techniques. ("NOER" denotes *no* edge ordering, otherwise edges are reordered by default.)

multiplies is dense. We analyze only the data cache that is closest to the main memory (e.g., secondary or L2 cache, in case of a two-level cache hierarchy).

In the noninterlaced case, the resulting matrix is of very wide bandwidth close to N. This makes the working set of the matrix-vector product operation in the cache close to N/W_{sc}, where W_{sc} is the cache block size in double words. If C_{sc} is the cache capacity in double words, then the number of conflict cache misses is bounded by

$$N\left\lceil\frac{N-C_{sc}}{W_{sc}}\right\rceil \tag{1}$$

when $N \geq C_{sc}$.

In the interlaced case, the unknowns at a grid point are stored together. With a good node reordering strategy, the matrix resulting out of some discretization of a PDE can be made to have a narrow bandwidth, β, which is much smaller than N. This results in the fewer conflict misses, bounded by the following expression:

$$N\left\lceil\frac{\beta-C_{sc}}{W_{sc}}\right\rceil \tag{2}$$

when $\beta \geq C_{sc}$.

We can derive similar expressions for the bounds (1 and 2) on TLB misses where C_{sc} will be replaced by the number page table entries (PTE), C_{TLB} and W_{sc} by the the memory page size, W_{mem}. Since the interlaced storage works on the data items closely spaced in memory, it causes fewer TLB misses as compared to those in the noninterlaced case.

2.1.2 Structural Blocking

Once the field data is interlaced, it is natural to use a block storage format for the Jacobian matrix of a multicomponent system of PDEs. The block size is the number of components (unknowns) per mesh point. As shown for the sparse matrix-vector case in [10], this structural blocking significantly reduces the number of integer loads and enhances the reuse of the data items in registers. It also reduces the required memory bandwidth for optimal performance.

Table 2: Execution times on a 250 MHz Origin 2000 for 357,900 vertex case with single or double precision storage of the preconditioner matrix. The results suggest that the linear solver time is bottlenecked by memory bandwidth. This conclusion is supported by analytical estimates in [10].

Number of Processors	Computational Phase			
	Linear Solve		Overall	
	Double	Single	Double	Single
16	223s	136s	746s	657s
32	117s	67s	373s	331s
64	60s	34s	205s	181s
120	31s	16s	122s	106s

2.1.3 Edge and Node Reorderings

In the original FUN3D code, the edges are colored for good vector performance. No pair of nodes in the same discretization stencil shares a color. This results in a very low cache line reuse. In addition, since consecutive memory references may be far apart, the TLB misses are a grave concern. About 70% of the execution time is spent serving TLB misses. As shown in Figure 3, this problem is effectively addressed by reordering the edges.

The edge reordering we have used is obtained by sorting the edges in increasing order by the node number at the one end of each edge. In effect, this converts an edge-based loop into a vertex-based loop that reuses vertex-based data items in most or all of the stencils that reference them several times before discarding it. And since a loop over edges goes over a node's neighbors first, this (in conjunction with a bandwidth reducing ordering for nodes) results in memory references that are closely spaced. Hence, the number of TLB misses is reduced significantly. For vertex ordering, we have used the Reverse Cuthill McKee (RCM) [8], which is known for reducing cache misses by creating more spatial locality.

2.2 Reducing the Required Memory Bandwidth

The CFD application spends almost all of its time in two phases: flux computations, to evaluate conservation law residuals, and sparse linear algebraic kernels, to solve the Newton equations for an iterative correction to the solution. The linear algebraic kernels run at close to the aggregate memory-bandwidth limit on performance (as determined by the STREAM benchmarks [17]), and the flux computations are bounded by instruction scheduling, that is, the number of basic operations that can be performed in a single clock cycle (see the analysis in [10]).

To improve the performance of the sparse triangular matrix solution phase (and of other similar phases where memory bandwidth is a bottleneck), we store elements of the preconditioner for the Jacobian matrix in single precision. In our "matrix-free" implementation, the Jacobian itself is never explicitly needed; see [11]. All computation with the preconditioner is still done in full (double) precision. The performance advantages are shown in Table 2, where the single precision storage version runs at almost twice the rate of the double precision storage version, clearly identifying memory bandwidth as the bottleneck. The percentage of overall time in the linear solver ranges from 29.9% to 25.4% for the double precision storage version to 20.7% to 15.1% for the single. The number of time steps needed to converge is not affected, since the preconditioner is already very approximate by design.

2.3 Parallel Issues

The traditional factors that limit scalability include global reductions, global data exchange, nearest neighbor data exchange, and the wait times at communication events due to load imbalance. We will call these wait times “implicit synchronizations”. Unfortunately, reducing the implicit synchronization by better load balancing can degrade the algorithmic convergence rate (discussed below), resulting in longer execution time. For grid-based PDE solvers, the global data exchange is seldom needed and is not considered in our work.

2.3.1 Scalability Bottlenecks

In Table 3, we present a closer look at the performance scalability results on the ASCI Red machine from 128 to 1024 nodes. The focus here is to identify the factors that retard the scalability. The overall parallel efficiency (denoted by $\eta_{overall}$) is broken into two components: (1) η_{alg} measures the degradation in the parallel efficiency due to the increased iteration count of this (non-coarse-grid-enhanced) NKS algorithm as the number of subdomains increases, while (2) η_{impl} measures the degradation coming from all other nonscalable factors such as global reductions, load imbalance (implicit synchronizations), and hardware limitations.

From Table 3, we observe that the global reductions for these runs are harmless. The primary factors responsible for the degradation of η_{impl} are the implicit synchronizations and to some degree the ghost point updates (scatters). We note that simply removing some of the synchronization points in the algorithm (for example, the inner products), will not significantly decrease the time spent on the implicit synchronizations, since the load imbalance will still exist and the wait times will shift to the next communication event in the code.

Interestingly, the increase in the percentage of time (3% to 6%) for the scatters results more from algorithmic issues than from hardware/software limitations. With an increase in the number of subdomains, the percentage of grid point data that must be communicated also rises. For example, the total amount of nearest neighbor data that must be communicated per iteration for 128 processors is 2 gigabytes, while for 1024 processors it is 5.3 gigabytes. Though more network wires are available when more processors are employed, this helps account for the increase in scatter times (from 3% to 6%). The final column in Table 3 shows the scalability of the “application level effective bandwidth” that is computed by dividing the total amount of data transferred by the time spent in scatter operation. It includes the message packing and unpacking times plus any contention in the communication. That is why it is far lower than the achievable bandwidth of the networking hardware.

2.3.2 Effect of Partitioning Strategy

Mesh partitioning has a dominant effect on parallel scalability for problems characterized by (almost) constant work per point. Poor load balance causes idleness at synchronization points, which are frequent in implicit methods (e.g., at every conjugation step in a Krylov solver). It is natural, then, with NKS methods to strive for a very well balanced load. The p-MeTiS algorithm in the MeTiS package [13], for example, provides almost perfect balancing of the number of mesh points per processor.

Figure 4 shows the effect of data partitioning using p-MeTiS, which tries to balance the number of nodes and edges on each partition, and k-MeTiS, which tries to reduce the number of noncontiguous subdomains and connectivity of the subdomains. We see that better overall scalability is observed with k-MeTiS, despite the better load balance for the p-MeTiS partitions. This is due to the slightly poorer numerical convergence rate of the iterative NKS algorithm with the p-MeTiS partitions. The poorer convergence rate can be explained by the fact that the p-MeTiS partitioner generates disconnected pieces within a single subdomain, effectively increasing the number of blocks in the block Jacobi or additive Schwarz algorithm. The convergence rates for block iterative methods degrade with increasing number of blocks [7, 19].

Table 3: Scalability bottlenecks on ASCI Red for the 2.8 million vertex mesh. The preconditioner used in these results is block Jacobi with ILU(1) in each subdomain (in Figure 1, block Jacobi with ILU(0) was used). We observe that the principal nonscaling factors are the "implicit synchronizations" and the scatters used for nearest neighbor communication.

Number of Processors	Its	Time	Speedup	Efficiency		
				$\eta_{overall}$	η_{alg}	η_{impl}
128	22	2,039s	1.00	1.00	1.00	1.00
256	24	1,144s	1.78	0.89	0.92	0.97
512	26	638s	3.20	0.80	0.85	0.94
768	27	441s	4.62	0.77	0.81	0.95
1024	29	362s	5.63	0.70	0.76	0.93

	Percent Times for			Scatter Scalability	
Number of Processors	Global Reduc-tions	Implicit Synchro-nizations	Ghost Point Scatters	Total Data Sent per Iteration (GB)	Application Level Effective Bandwidth per Node (MB/s)
128	5	4	3	2.0	3.9
256	3	6	4	2.8	4.2
512	3	7	5	4.0	3.4
768	3	8	5	4.6	4.2
1024	3	10	6	5.3	4.2

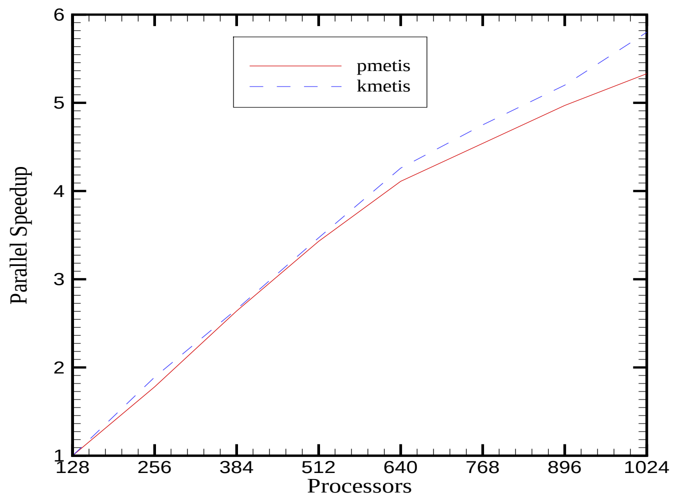


Figure 4: Parallel speedup relative to 128 processors on a 600 MHz Cray T3E for the 2.8 million vertex case, showing the effect of partitioning algorithms k-MeTiS, and p-MeTiS. The former is a superior strategy when the number of subdomains becomes very large.

2.4 Algorithmic Tuning for ψNKS Solver

The following is an incomplete list of parameters that need to be tuned in various phases of a pseudo-transient Newton-Krylov-Schwarz algorithm.

- Nonlinear robustness continuation parameters: discretization order, initial timestep, exponent of timestep evolution law (see below)
- Newton parameters: convergence tolerance on each time step, globalization strategy (line search or trust region parameters), refresh frequency for Jacobian preconditioner
- Krylov parameters: convergence tolerance for each Newton correction, restart dimension of Krylov subspace, overall Krylov iteration limit, orthogonalization mechanism
- Schwarz parameters: subdomain number, quality of subdomain solver (fill level, number of sweeps), amount of subdomain overlap, coarse grid usage
- Subproblem parameters: fill level, number of sweeps

We highlight a few of these tunings that have yielded good results in our context. Some subsets of these parameters interact with each other. In addition, optimal values of some of these parameters depend on the grid resolution. Our strategy has been first to find a suitable range of variation for the parameter under consideration. In all of our experiments, the goal has been to minimize the overall execution time, not to maximize the floating-point operations per second. There are many tradeoffs that enhance Mflop/s rates but retard execution completion.

2.4.1 Parameters for Pseudo Transient Continuation

Although asymptotically superlinear, solution strategies based on Newton's method must often be approached through pseudo-timestepping. For robustness, pseudo-timestepping is often initiated with very small timesteps and accelerated subsequently. However, this conventional approach can lead to long "induction" periods that may be bypassed by a more aggressive strategy, especially for the smooth flow fields.

The timestep is advanced toward infinity by a power-law variation of the switched evolution/relaxation (SER) heuristic of Van Leer and Mulder [18]. To be specific, within each residual reduction phase of computation, we adjust the timestep according to

$$N^{\ell}_{CFL} = N^{0}_{CFL} \left(\frac{||f(u^0)||}{||f(u^{\ell-1})||} \right)^p ,$$

where p is a tunable exponent close to unity. Figure 5 shows the effect of initial CFL number (the Courant-Friedrich-Levy number, a dimensionless measure of the timestep size), N^{0}_{CFL}, on the convergence rate. In general, the best choice of initial CFL number is dependent on the grid size and Mach number. A small CFL adds nonlinear stability far from the solution but retards the approach to the domain of superlinear convergence of the steady state. For flows with near discontinuities, it is safer to start with small CFL numbers.

In flows with shocks, high-order (second or higher) discretization for the convection terms should be activated only after the shock position has settled down. We begin such simulations with a first-order upwind scheme and switch to second-order after a certain residual reduction. The exponent (p) in the power law above is damped to 0.75 for robustness when shocks are expected to appear in second-order discretizations. For first-order discretizations, this exponent may be as large as 1.5. A reasonable switchover point of the residual norm between first-order and second-order discretization phases has been determined empirically. In shock-free simulations we use second-order accuracy throughout. Otherwise, we normally reduce the first two to four orders of residual norm with the first-order discretization, then switch to second. This order of accuracy applies to the flux calculation. The preconditioner matrix is always built out of a first-order analytical Jacobian matrix.

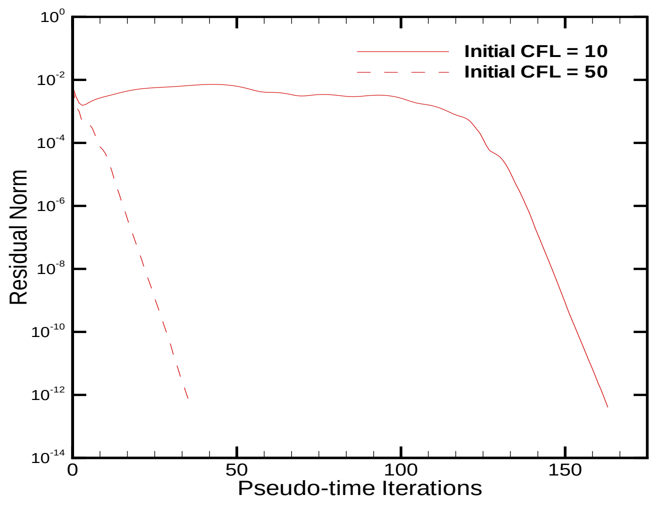


Figure 5: Residual norm versus iteration count for 2.8 million vertex case, showing the effect of initial CFL number on convergence rate. The convergence tuning of nonlinear problems is notoriously case specific.

2.4.2 Parameters for Krylov Solver

We use an inexact Newton method on each timestep [9]; that is, the linear system within each Newton iteration is solved only approximately. Especially in the beginning of the solution process, this saves a significant amount of execution time. We have considered the following three parameters in this phase of computation: convergence tolerance, the number of simultaneously storable Krylov vectors, and the total number of Krylov iterations. The typical range of variation for the inner convergence tolerance is 0.001–0.01. We have experimented with progressively tighter tolerances near convergence, and saved Newton iterations thereby, but did not save time relative to cases with loose and constant tolerance. The Krylov subspace dimension depends largely on the problem size and the available memory. We have used values in the range of 10–30 for most of the problems. The total number of linear iterations (within each nonlinear solve) has been varied from 10 for the smallest problem to 80 for the largest one. A typical number of fine-grid flux evaluations for achieving 10^{-10} residual reduction on a million-vertex Euler problem is a couple of thousand.

2.4.3 Additive Schwarz Preconditioner

The Krylov method needs to be preconditioned for acceptable inner iteration convergence rates, and the preconditioning can be the “make-or-break” feature of an implicit code. A good preconditioner saves time and space by permitting fewer iterations in the Krylov loop and smaller storage for the Krylov subspace. An additive Schwarz method (ASM) preconditioner [6] accomplishes this in a concurrent, localized manner, building an approximate inverse to the global Jacobian out of approximate solutions within each subdomain of a partitioning of the global PDE domain.

Schwarz methods are distinguished by the amount of overlap in the partitions governing each subdomain and by the presence or absence of a coarse grid spanning the independent subdomains. A beautiful theory exists [19] quantifying the convergence gains of overlap and the use of multiple levels, which have to be balanced against the cost of these operations on real parallel computer hardware.

In Table 4, we show some runs of PETSc-FUN3D on the ASCI Red machine for a fixed-size problem with 357,900 grid points and 1,789,500 degrees of freedom. These calculations were performed using GMRES(20), one subdomain per processor (without overlap for block Jacobi and with overlap for ASM), and ILU(k) where k varies from 0 to 2, and with the natural ordering in each subdomain block. The use of ILU(0) with natural ordering on the first-order Jacobian, while applying a second-order operator, allows the factorization to be done in place, with or without overlap. However, the overlap case does require forming an additional data structure on each processor to store matrix elements corresponding to the overlapped regions.

From Table 4 we see that the larger overlap and more fill helps in reducing the total number of linear iterations as the number of processors increases, as theory and intuition predict. However, both increases consume more memory, and both result in more work per iteration, ultimately driving up execution times in spite of faster convergence. Best execution times are obtained for any given number of processors for ILU(1), as the number of processors becomes large (subdomain size small), for zero overlap.

The additional computation/communication costs for the ASM (as compared with block Jacobi) are

1. Calculation of the matrix couplings among processors. For block Jacobi, these never need to be calculated. However, the code always computes them currently.
2. Communication of the "overlapped" matrix elements to the relevant processors.
3. Factorization of the larger local submatrices.
4. Communication of the ghost points in the application of the ASM preconditioner. We use restricted additive Schwarz (RASM) [7], which has only one communication phase while applying the preconditioner, as opposed to two for standard ASM.
5. The larger triangular solves required for each iteration.

We note that the execution times as reported in Table 4 are highly dependent on the machine used, since each of the additional computation/communication costs (as mentioned above) may interact differently (depending on architectural parameters such as memory bandwidth, cache size, and the communication network etc.).

2.5 Domain-based and/or Instruction-level Parallelism

The performance results presented so far were based on the domain-based parallelism using the Message Passing Interface (MPI). With the availability of large scale SMP clusters, the different software models for parallel programming require a fresh assessment. For machines with physically distributed memory, MPI has been a natural and very successful software model. For another category of machines with distributed shared memory and nonuniform memory access, both MPI and OpenMP have been used with respectable parallel scalability. But for clusters with two or more SMPs on a single node, the mixed software model with threads within a node (OpenMP being a special case of threads because of the potential for highly efficient handling of the threads and memory by the compiler) and MPI between the nodes looks quite natural. Several researchers (e.g., [5, 16]) have used this mixed model with reasonable success.

We investigated the mixed model by employing OpenMP only in the flux calculation phase. This phase takes over 60% of the execution time and is an ideal candidate for shared-memory parallelism because it does not suffer from the memory bandwidth bottleneck. In Table 5, we compare the performance of this phase when the work is divided by using two OpenMP threads per node with the performance when the work is divided using two independent MPI processes per node. We note that there is no communication in this phase. Both processors work with the same amount of memory available on a node; in the OpenMP case, it is shared between the two threads, while in the case of MPI it is divided into two address spaces.

The hybrid MPI/OpenMP programming model appears to be a more efficient way to employ shared memory than are the heavyweight subdomain-based processes (MPI alone), especially when the number of nodes is large. The MPI model works with larger number of subdomains (equal to the number of MPI processors),

Table 4: Execution times and linear iteration counts on the 333 MHz Pentium Pro ASCI Red machine for the 357,900K vertex case, showing the effect of subdomain overlap and incomplete factorization fill level in the Additive Schwarz preconditioner. **The best execution times for each ILU fill level and number of processors are in boldface in each row.**

ILU(0) in Each Subdomain						
Number of Processors	Overlap					
	0		1		2	
	Time	Linear Its	Time	Linear Its	Time	Linear Its
32	688s	930	**661s**	816	696s	813
64	**371s**	993	374s	876	418s	887
128	**210s**	1052	230s	988	222s	872

ILU(1) in Each Subdomain						
Number of Processors	Overlap					
	0		1		2	
	Time	Linear Its	Time	Linear Its	Time	Linear Its
32	598s	674	**564s**	549	617s	532
64	**334s**	746	335s	617	359s	551
128	**177s**	807	178s	630	200s	555

ILU(2) in Each Subdomain						
Number of Processors	Overlap					
	0		1		2	
	Time	Linear Its	Time	Linear Its	Time	Linear Its
32	**688s**	527	786s	441	—-	—
64	**386s**	608	441s	488	531s	448
128	**193s**	631	272s	540	313s	472

resulting in slower rate of convergence. The hybrid model works with fewer chunkier subdomains (equal to the number of nodes) that result in faster convergence rate and shorter execution time, despite the fact that there is some redundant work when the data from the two threads is combined due to the lack of a vector-reduce operation in the OpenMP standard (version 1) itself. Specifically, some redundant work arrays must be allocated that are not present in the MPI code. The subsequent gather operations (which tend to be memory bandwidth bound) can easily offset the advantages accruing from the low latency shared memory communication. One way to get around this problem is to use some coloring strategies to create the disjoint work sets, but this takes away the ease and simplicity of the parallelization step promised by OpenMP model.

3 Conclusions

Unstructured implicit CFD solvers are amenable to scalable implementation, but careful tuning is needed to obtain the best product of per-processor efficiency and parallel efficiency. The number of cache misses and the achievable memory bandwidth are two important parameters that should be considered to determine an optimal data storage pattern. We have introduced simple analytical models to predict the bounds on cache misses for an important kernel, sparse matrix-vector product. The impact of data reorganizing strategies (interlacing, blocking, and edge/vertex reorderings) is demonstrated through the sparse matrix-vector product model and hardware counter profiling.

We are working on more detailed performance models that will help us to predict more realistic (or achievable) performance bounds [10] for the PETSc-FUN3D code (and other similar codes) than are avail-

Table 5: Execution time on the 333 MHz Pentium Pro ASCI Red machine for function evaluations only for the 2.8 million vertex case, showing differences in exploiting the second processor sharing the same memory with both OpenMP instruction-level parallelism (number of subdomains equals the number of nodes) and MPI domain-level parallelism (number of subdomains is equal to the number of processes per node).

	MPI/OpenMP Threads per Node		MPI Processes per Node	
Nodes	1	2	1	2
256	483s	261s	456s	258s
2560	76s	39s	72s	45s
3072	66s	33s	62s	40s

able today. Surprisingly, on reasonably well-balanced parallel machines (e.g. ASCI Red and Cray T3E), the parallel scalability (for several thousand nodes) for the NKS algorithm is limited by the algorithmic requirements of increased nearest neighbor communication (because a higher fraction of degrees of freedom are shared by multiple processors) and implicit synchronizations induced by subdomain load imbalance. Preliminary investigation of the hybrid (MPI/OpenMP) approach indicates the potential of superior performance for parts of the code that are not memory bandwidth limited. In the future, we will further examine these issues from a memory centric perspective.

Acknowledgments

Collaborations with Lois C. McInnes, Satish Balay, W. Kyle Anderson, and Dimitri Mavriplis were critical to the work leading up to this paper. Debbie Swider's assistance with many of the ASCI platform runs is gratefully acknowledged. Computer time was supplied by Argonne National Laboratory, Lawrence Livermore National Laboratory, National Energy Research Scientific Computing Center (NERSC), Sandia National Laboratories, and SGI-Cray.

References

[1] W. K. Anderson and D. L. Bonhaus. An implicit upwind algorithm for computing turbulent flows on unstructured grids. *Computers and Fluids*, 23:1–21, 1994.

[2] W. K. Anderson, W. D. Gropp, D. K. Kaushik D. E. Keyes, and B. F. Smith. Achieving high sustained performance in an unstructured mesh CFD application. In *Proceedings of SC'99*, 1999. Gordon Bell Prize Award Paper in Special Category.

[3] W. K. Anderson, R. D. Rausch, and D. L. Bonhaus. Implicit/multigrid algorithms for incompressible turbulent flows on unstructured grids. *J. Computational Physics*, 128:391–408, 1996.

[4] Satish Balay, William Gropp, Lois Curfman McInnes, and Barry Smith. The Portable Extensible Toolkit for Scientific Computing (PETSc) version 28. `http://www.mcs.anl.gov/petsc/petsc.html`, 2000.

[5] S. W. Bova, C. P. Breshears, C. E. Cuicchi, Z. Demirbilek, and H. A. Gabb. Dual-level parallel analysis of harbor wave response using MPI and OpenMP. *High Performance Computing Applications*, 14:49–64, 2000.

[6] X. C. Cai. Some domain decomposition algorithms for nonselfadjoint elliptic and parabolic partial differential equations. Technical Report 461, Courant Institute, New York, 1989.

[7] X. C. Cai and M. Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM J. Scientific Computing*, 21:792–797, 1999.

[8] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *ACM Proceedings of the 24th National Conference*, 1969.

[9] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact Newton methods. *SIAM J. Numerical Analysis*, 19:400–408, 1982.

[10] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Toward realistic performance bounds for implicit CFD codes. In D. Keyes, A. Ecer, J. Periaux, N. Satofuka, and P. Fox, editors, *Proceedings of Parallel CFD'99*, pages 233–240. Elsevier, 1999.

[11] W. D. Gropp, L. C. McInnes, M. D. Tidriri, and D. E. Keyes. Globalized Newton-Krylov-Schwarz algorithms and software for parallel implicit CFD. *Int. J. High Performance Computing Applications*, 14:102–136, 2000.

[12] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[13] G. Karypis and V. Kumar. A fast and high quality scheme for partitioning irregular graphs. *SIAM J. Scientific Computing*, 20:359–392, 1999.

[14] D. K. Kaushik, D. E. Keyes, and B. F. Smith. Newton-Krylov-Schwarz methods for aerodynamic problems: Compressible and incompressible flows on unstructured grids. In C.-H. Lai et al., editors, *Proceedings of the 11th International Conference on Domain Decomposition Methods*, pages 513–520. Domain Decomposition Press, Bergen, 1999.

[15] C. T. Kelley and D. E. Keyes. Convergence analysis of pseudo-transient continuation. *SIAM J. Numerical Analysis*, 35:508–523, 1998.

[16] D. J. Mavriplis. Parallel unstructured mesh analysis of high-lift configurations. Technical Report 2000-0923, AIAA, 2000.

[17] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, 1995. `http://www.cs.virginia.edu/stream`.

[18] W. Mulder and B. Van Leer. Experiments with implicit upwind methods for the Euler equations. *J. Computational Physics*, 59:232–246, May 1985.

[19] B. F. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition*. Cambridge University Press, 1996.